Wait-Free Algorithms for Heaps *

Greg Barnes

Dept. of Computer Science and Engineering, FR-35

University of Washington

Seattle, WA 98195

February 3, 1992

Abstract

This paper examines algorithms to implement heaps on shared memory multiprocessors. A natural model for these machines is an asynchronous parallel machine, in which the processors are subject to arbitrary delays. On such machines, it is desirable for algorithms to be *wait-free*, meaning that each thread makes progress independent of the other threads executing on the machine. We present a wait-free algorithm to implement heaps. The algorithms are similar to the general approach given in [4], with optimizations that allow many threads to work on the heap simultaneously, while still guaranteeing a strong serializability property.

1 Introduction

We are interested in designing efficient data structures and algorithms for shared memory multiprocessors. Processors on these machines may execute instructions at a varying rate (due to cache behavior, for example), and are subject to long delays (e.g. when swapped out by the scheduler, or after a page fault). Programs are executed by a collection of *threads*, which are time-shared among the processors. There may be more threads than processors, so the user can view a program as running on an arbitrarily large collection

*E-mail address: `greg@cs.washington.edu`. Research Supported by NSF Grant CCR-9002891.

of processors subject to arbitrary delays. A natural model to capture this behavior is the asynchronous parallel machine, where the processors can suffers delays of any length at any time. In such a model, it is desirable for algorithms to be wait-free. This paper presents a wait-free algorithm to manipulate heaps. The algorithm uses an approach significantly different from previous wait-free algorithms for concurrent objects.

The only previously known wait-free algorithm to manipulate heaps arises from Herlihy's work on a methodology for implementing concurrent objects [15]. Herlihy's basic methodology requires a thread to check out a pointer to the object, make and change a copy of the object, and check the object back in. If another thread has changed the pointer variable since it was checked out, the check-in fails and the thread must start over. The *copying algorithm* for heaps is a more efficient variant of this procedure that copies only $O(\log n)$ of the n nodes per operation, instead of the entire heap.

Our algorithm achieves theoretical performance similar to the copying algorithm. Its upper and lower bounds are asymptotically the same, and it can give similar serializability guarantees on its output. Its approach, however, is very different. In contrast to the copying algorithm, our algorithm maintains only one copy of the heap data structure. Threads complete heap operations by making a series of small changes to this structure. For this algorithm to work correctly, it is necessary that threads be able to detect and complete an incomplete series of changes to the structure. Bershad describes a similar idea for implementing synchronization primitives on machines that do not support these primitives [6, 7].

This approach offers four advantages. First, we expect the algorithm to achieve better speedup than the copying algorithm. The copying algorithm must complete one entire heap operation before another can begin, while this algorithm allows a new operation to begin after an operation has completed a constant number of steps. Second, our algorithm maintains the random access nature of the heap by storing it in an array. The copying algorithm only copies $O(\log n)$ nodes if the heap is stored as a tree using pointers. Third, we expect the algorithm to repeat less work, because it spreads threads across the entire structure instead of concentrating them on one pointer. In certain pathological cases, our algorithm will repeat as much work as the copying algorithm, because the threads will all work in the same section in the tree. In most cases, however, we expect the threads will spread out evenly as they descend the tree. There will be less contention and less unnecessary work the farther a thread progresses away from the root, and therefore less work overall. Finally, this approach offers hope for more efficient wait-free implementations of certain data structures. The only known algorithm for most structures is a copying algorithm derived from Herlihy's methodology, but it is difficult to see how a copying algorithm for linear or array-based objects can avoid copying the entire structure. An algorithm similar to ours would be more desirable, since it would change only those portions of the object that are affected by a given operation.

2 Background and Previous Work

Typically, concurrent access to shared objects is implemented using critical sections guarded by locks. Locks are an unattractive solution in asynchronous models, because the thread holding a lock can be delayed indefinitely, forcing the other threads to wait uselessly until the lock is released. It is more desirable if algorithms are *non-blocking*, that is, they always guarantee at least one thread will complete an operation in a finite number of steps. A *wait-free* algorithm is a special case of a non-blocking algorithm that guarantees *all* threads will complete their work in a finite number of steps.

Early work on wait-free objects focused on proving the power of various synchronization primitives. Herlihy [16] unifies much of this work by showing the existence of *universal* primitives, such as Compare&Swap, which can be used to implement any wait-free object. Using Load_Linked and Store_Conditional, a universal pair of primitives similar to Compare&Swap, Herlihy [15] describes a methodology for converting synchronous implementations of data structure algorithms to non-blocking and wait-free implementations. Alemany and Felten [1] present techniques for improving the performance of Herlihy's protocol in practice. Anderson and Woll [2] use Compare&Swap to design efficient asynchronous algorithms for the Union-Find problem.

Heaps are often used to implement priority queues. Many researchers have examined the problem of concurrent access to priority queue structures such as skew trees, B-trees, or 2-3 trees. Most existing algorithms use locks to control concurrent access to the structure [5, 8, 11, 13, 18, 22, 26].

Herlihy [15] provides the only previously known wait-free implementation of heaps. His basic methodology was explained above, in Section 1: make a copy of the object, change it, and try to check in the new copy, using the synchronization primitives to test whether the variable has been changed in the interim. Since copying the entire object can be time-consuming, Herlihy suggests that for large objects the programmer supply a copying algorithm that copies as little of the structure as possible. If the heap is stored as a tree using pointers, then for a heap with n nodes, a copying algorithm need only make new copies of $O(\log n)$ of the nodes per operation, the nodes that are changed by the operation, plus their ancestors.[1] Usually, heaps are stored in an array, thus allowing random access to the elements, but we see no way for a copying algorithm to maintain this constant time access without making a new copy of the array for every operation.

Our algorithm uses the Load_Linked and Store_Conditional synchronization primitives. Load_Linked acts like a load instruction. Store_Conditional is similar to a store instruction, but it succeeds only if no other thread has written the variable since the Load_Linked instruction. Store_Conditional returns a boolean value indicating whether the write succeeded or failed. Load_Linked and Store_Conditional can be efficiently implemented given a cache-coherent architecture, and are supported in the MIPS-II architecture [24].

The remainder of the paper is organized as follows. We begin with a discussion of the basic approach of the algorithm in Section 3. Section 4 presents a high-level sketch of the algorithm, and Section 5 provides a sketch of the proof of the correctness of the algorithm. Section 6 discusses the performance of the algorithm, and Section 7 conclude with some notes and suggestions for future work. A detailed presentation of the code and a proof of its correctness are omitted in this abstract.

3 Approach

A heap is a balanced binary tree. Each node in the tree has an associated *key*. Without loss of generality, we assume that all keys are unique. These keys obey the *heap property*: the parent's key is less than the keys of its children. The insert operation adds a key to the heap, and the delete_min operation removes the minimum key from the heap.

[1]Herlihy does not explicitly present this algorithm in his paper, but he does describe algorithms for heaps and skew heaps, and this algorithm is easily derived from his work.

Our algorithm stores the heap as an array of pointers to nodes. Each node is a record consisting of the key, some flags, and a few auxiliary variables. By using Load_Linked and Store_Conditional on the pointer to a record, the algorithm can atomically check in the entire record. A similar strategy is used by Anderson and Woll [2] in their Union-Find algorithms.

In standard heap algorithms, both the delete_min and insert operations are composed of a series of *suboperations* — a new leaf is added to the heap, or the root key is removed and replaced by the key of one of the leaves, and a series of swaps is performed to restore the heap property to the structure. In the copying algorithm, these suboperations are transparent to all threads except the one performing the operation; individual threads execute the suboperations on their local copies of the heap, not on the shared copy. In our algorithm, the suboperations are all performed on one shared heap.

Allowing multiple simultaneous changes to the same data structure introduces some interference problems. Consider Figure 1, depicting the two types of problems that can arise if multiple operations are allowed on the same heap. Figure 1(a) shows an incomplete operation. Threads T_1 and T_2 have both begun delete_min operations, and are sifting the keys 17 and 15 down the tree, but T_1 has been delayed. If we naively choose to implement the standard heap algorithms, T_2 could examine the keys of its children, notice that its key is less than both, and decide to stop. This is clearly wrong, since key 15 belongs below keys 5 and 6. Note that a similar situation can arise if two threads try to sift keys *up* the tree.

Figure 1(b) shows an incomplete suboperation. As before, T_1 and T_2 are trying to complete delete_min operations, but this time T_1 was delayed in the middle of swapping keys 17 and 5. Even if T_2 could detect that something was amiss, it has no way of knowing the key that was being swapped with 17, and cannot recover from this situation.

Most of these difficulties would be eliminated if all suboperations executed atomically; incomplete suboperations would never occur, and incomplete operations, while not a trivial problem, are not difficult to handle. For example, to solve the problem in Figure 1(a), it is sufficient to store a flag with each node indicating whether the key obeys the heap property. When T_2 reads n_2, it will notice that key 17 may not obey the heap property, and realize it should not stop sifting its key. It cannot, however, simply wait for T_1 to move its key farther down — T_1 could be delayed for an indefinite amount of time. In our algorithm, T_2 executes T_1's next swap for it. When the suboperation

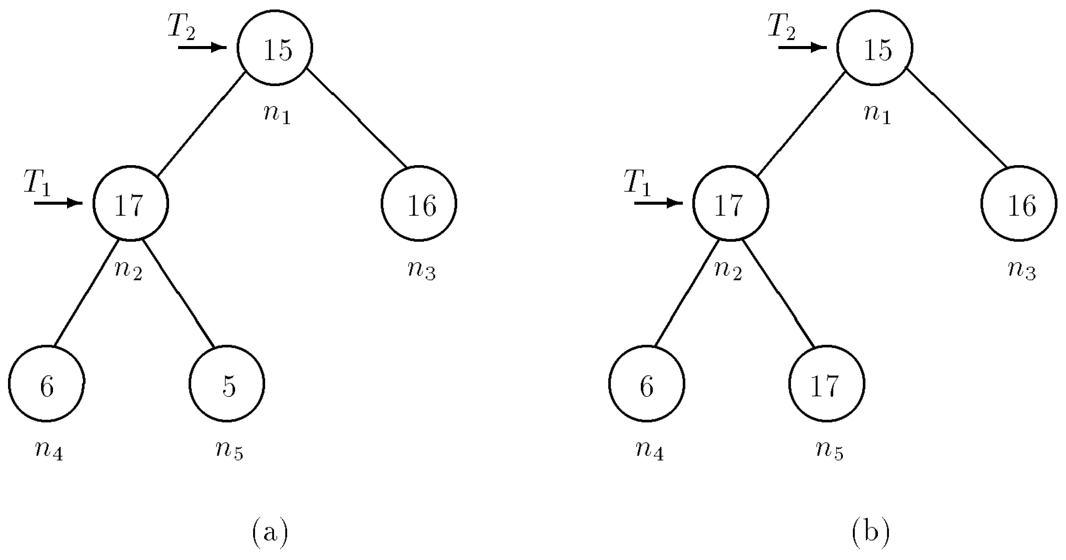

Figure 1: Two potential problems with multiple threads.

is completed, T_2 can continue with its own task.

Unfortunately, we do not know how to make suboperations atomic. Instead, our algorithm makes all suboperations *effectively atomic*.

Definition 3.1 *Let V_S be the set of variables changed by suboperation S, and say that S is officially begun when the first variable in V_S is changed by a thread trying to execute S. Then S is* effectively atomic *if, once it is begun, no other suboperation that changes a variable in V_S begins until S is completed.*

Guaranteeing effective atomicity makes it much easier to reason about the correctness of the algorithm, since once a suboperation is begun, it might as well have completed. To guarantee effective atomicity, the algorithm uses a strategy similar to the one used to solve the problem in Figure 1(a): a thread detects the suboperations that have not been completed by other threads, and completes them. For example, to solve the problem in Figure 1(b), T_1 stores not only the new key 17 in node n_5, but the old key 5. T_2 can detect the incomplete swap by reading n_5, and use the old key to complete the swap.

A thread indicates an incomplete suboperation by marking some or all of the variables it changes. Marking variables also helps delayed threads to recover when they are restarted. Before arguing that this strategy is sufficient, it is necessary to present some details about the algorithm.

4 The Algorithm

In this section, we first give a high-level sketch of the algorithms for the two operations, followed by a brief description of the main data structures and some important functions.

4.1 High-level sketch

To simplify the following discussion, we will assume the atomicity of suboperations. The algorithm supports two operations on the heap, delete_min and insert. delete_min takes one argument, a pointer to a location where the minimum key in the heap is to be stored. insert takes a key as its argument. Each operation is divided into a preliminary phase, in which a key is added to or deleted from the structure, and a sifting phase, in which any key that no longer obeys the heap property because of the preliminary phase is sifted to its proper location. Only one preliminary phase is executed at a time; before beginning a heap operation, a thread must successfully change a special record that holds all global variables associated with the heap, including the operation that is currently performing its preliminary phase. After a thread completes its preliminary phase, it changes the special variable so that another operation can begin, and begins its sifting phase.

Assuming it encounters no incomplete operations, a delete_min operation proceeds like the standard heap algorithm. The preliminary phase consists of three steps: the highest active leaf in the tree is marked as inactive, the key in this leaf is moved to the root, and the old root key is written in the desired location. In the sifting phase, the new root key is sifted down the tree by repeatedly swapping it with the lower-valued key among its children until it reaches a node where its key is less than its children's keys.

An insert operation also sifts a key down from the root, instead of up from a leaf as in the standard heap algorithm. The key is stored in a leaf, and is implicitly moved down the tree along the path from the root to this leaf.

Again assuming it encounters no incomplete operations, in the preliminary phase, the new key is stored in the lowest inactive leaf position, along with a pointer to the root. This pointer indicates the current location of the key in the sifting phase. The root is also modified to hold a corresponding pointer to the leaf. In the sifting phase, the leaf always points to one of its ancestors. If at any time, the key in the leaf is less than the key of the ancestor where it is pointing, the two keys are swapped. Otherwise, the appropriate pointers are changed to indicate the key has moved down one level in the sifting phase. We have developed a version of the algorithm that mimics the standard algorithm by sifting inserted keys up from a leaf, but sifting down from the root allows for clean serialization in the algorithm (see Section 5.1 below), and simplifies the proof of correctness somewhat, since all keys are sifted in the same direction.

4.2 Data Structures and Functions

The algorithm uses two main data structures, the `heap_status` record, which stores all global information about the heap, and the `heap_entry` record, which contains the information about a node. These records contain the usual heap variables, as well as extra variables and flags to indicate partially completed operations and suboperations. The `heap_status` record (the "special variable" mentioned in Section 4.1 above) contains the size of the heap, all information about the operation whose preliminary phase is currently being performed, including the value being inserted (for an insert operation) or the memory location where the minimum key should be written (for a delete_min), and a counter indicating the number of preliminary phases that have been performed. This counter provides a unique identifier for each operation. As mentioned before, the heap itself is stored in an array. Each entry in the array is a pointer to a `heap_entry` record, which contains a key, as well as a **status** field and some auxiliary variables. The **status** field is an enumerated type that can denote nodes that are being sifted down as the result of delete_min or insert operations, nodes that are being swapped, nodes that have no associated uncompleted work, inactive leaves, etc. There are three types of auxiliary variables in the `heap_entry` record — variables that hold old keys (used, for example, to recover from the swap problem in Figure 1), variables that point to other nodes (used in the sifting phase of an insert operation), and ID variables, which store the **opID** of an operation associated with this node. The ID variables help delayed processes detect their condition when they wake up.

Before any preliminary phase is begun, a thread must successfully change the heap's heap_status record to hold the information about its operation. The thread loads the record, and if the **op** field indicates another preliminary phase is in progress, the thread completes that phase and changes the record to indicate the heap is "free". The thread then tries to change the status (using the Store_Conditional instruction) to hold information about its operation; if the Store_Conditional fails, it must start over. Note that this is not a fair algorithm, as a thread could continually be prevented from beginning its operation if its timing is particularly bad. Still, assuming a finite number of operations are to be performed on the heap, this algorithm is wait-free. Given an unbounded number of operations, the algorithm as written is merely non-blocking, but an auxiliary scheduling scheme, such as the one described by Herlihy [15], would solve this problem. It is not difficult to show that this process guarantees that no more than one preliminary phase will be executed at a time.

The most important function in the algorithm is fix. A thread calls fix before attempting to change any node. fix returns a *clean* copy of the node — it examines the node's status, and finishes any uncompleted work associated with it.

5 Correctness

The following paragraphs sketch the main ideas behind the proof of the algorithm's correctness. The key to proving the correctness of the algorithm is to prove that it guarantees the effective atomicity of suboperations. The key to proving atomicity is to show that an incomplete suboperation can be detected before another suboperation that changes one of its variables is begun — once it is detected, it is not difficult to see that a thread can finish a suboperation begun by another thread using the data structures described in Section 4.2. In Section 5.1 we will argue that given effectively atomic suboperations, the algorithm performs like a sequential algorithm (in particular, that a delete_min operation always gives the minimum key in the heap).

Assuming all previous suboperations were effectively atomic, a thread, T, that begins a new suboperation will not have trouble with threads executing older suboperations. Any old suboperation that changes the same variables must be completed before T's suboperation can begin, and any thread still executing such an suboperation will discover the suboperation has completed as soon as it reads or tries to Store_Conditional a variable.

The only other possible interference comes from threads that try to begin suboperations after T's suboperation has begun. T can always mark the variables it changes to guard against interference from these threads, because all operations and suboperations follow basic patterns. Both operations follow a similar procedure — the root and a leaf are changed, and a key is sifted down the tree. There are four suboperations: the preliminary phases of the two operations, the swap in the sifting phase of a delete_min, and the implicit move down in the sifting phase of an insert (the algorithm combines the possible swap with the move down into one suboperation). All these suboperations operate on a similar set of nodes (an ancestor node, and a child of the ancestor and/or one of its descendants). Therefore, T need only guard against two possibilities: a thread that comes down from above, and a thread that changes a leaf from below. The ancestor node guards the suboperation from above, either because the ancestor itself is marked (as in sifting operations, where the node's **status** indicates an incomplete operation), or because there is nothing above it (as in preliminary phases, when the ancestor is the root). To guard against threads that change leaves, T begins by changing the lowest descendant in the set of variables its suboperation changes. Once the descendant is successfully changed, any thread that tries to begin a new suboperation that changes one of the same variables will have to read either the ancestor or the descendant node before it begins, and will be able to detect if T's suboperation is incomplete.

5.1 Serializability of Operations

The standard notion of correctness in asynchronous parallel algorithms is to assume the atomic instructions of all threads are interleaved in some linear order; the algorithm is correct if it behaves properly for all such interleavings [17, 21]. In this context, proper behavior is defined by the results of the delete_min operations; if the results of the delete_min operations correspond to some serialization of the operations, the algorithm is correct. We can make a stronger assertion.

Theorem 5.1 *Let $Q = op_1, op_2, \ldots op_K$ be the operations performed by the algorithm on the heap, in the order that the operations successfully execute their preliminary phases. Then the results of the* delete_min *operations in this asynchronous sequence are the same as the results returned by a uniprocessor heap algorithm given the sequence of operations, Q.*

The theorem is proved using the following lemmas.

Lemma 5.2 *Any clean node always has a lower-valued key than any of its clean descendants.*

The proof proceeds by induction, based on the fact that none of the suboperations, if executed atomically, will make the property false if it was true before.

Lemma 5.3 *A* delete_min *operation will always return the minimum key in the tree.*

Proof:[sketch] delete_min always calls fix to get a clean copy of the root. After fix returns, it is possible that all currently uncompleted sifting phases could be completed before the root's key is deleted. The resulting tree would have the same key at the root and hold the same keys, and all its nodes would be clean. Since this is a valid interleaving of instructions, by Lemma 5.2, the key in the root must be less than all other keys in the tree. □

Note that Lemma 5.3 would not be true if inserted keys were sifted up from the leaves.

6 Performance

In recent years, researchers have proposed many different versions of the asynchronous PRAM, or APRAM (including [9, 10, 12, 25]), most with differing notions of run-time. We measure the performance of our algorithm using *work*, the same measure used in a series of papers on fault-tolerant PRAMs [20, 19, 23]. The work done by an algorithm is the total number of steps taken by all threads.

In the absence of other threads, every operation takes $O(\log n)$ work, where n is the maximum number of nodes in the tree. The total amount of work used to perform K operations, then, is $O(K \log n)$ plus the amount of work expended unnecessarily because of multiple threads (two threads trying to perform the same operation at the same time, an unsuccessful Store_Conditional, etc.). The following lemma gives an upper bound on the performance of the algorithm, including this unnecessary work.

Lemma 6.1 *The algorithm uses no more than $O(Kp \log n)$ work to perform K heap operations, where n is the maximum number of nodes in the tree and p the number of threads.*

Proof:[sketch] Amortize the work by charging a suboperation for work done by any thread trying to complete the suboperation. A thread only performs constant work on a suboperation, and each operation consists of $O(\log n)$ suboperations, so the total work is $O(Kp \log n)$. □

Unfortunately, this bound is not very good, since at best this implies no speedup at all over the sequential case. Even worse, given a strong adversary that can impose any ordering on the atomic operations of the threads, this bound is tight.

Lemma 6.2 *The algorithm can use $\Omega(Kp \log n)$ time to perform K heap operations.*

One scenario where the adversary can force this much work is a tree that has a path from the root to a leaf where none of the nodes obey the heap property. If $\Omega(p)$ threads all try to fix the root, and they operate essentially in lock step, they could end up doing $\Omega(p \log n)$ work to complete $\Omega(\log n)$ suboperations. The adversary can recreate this situation repeatedly, yielding $\Omega(Kp \log n)$ total work.

These bounds are the same as those for the copying algorithm. Intuitively, the lower bound for both algorithms arises when most of the p threads work in the same position in the tree most of the time. In practice, we expect the threads in our algorithm will not concentrate on one area in the tree but will spread themselves evenly as they travel down the tree. In particular, a key being sifted down as a result of a delete_min operation should swap keys with the right child about as often as the left child. (The same cannot be said of insert sifting, since it follows a predefined path to a leaf, and successive insertions take place at neighboring leaves.) If this is true, the amount of work lost due to interfering delete_min operations will decrease by at least a factor of two for every level the node is from the root — about half the keys at a node will be sifted down to each child, and some keys will not be sifted at all. Informally, since both algorithms should suffer the same contention at the root, this means our algorithm will spend a factor of $O(\log n)$ less time than the copying algorithm on lost work due to delete_min operations.

We also expect the algorithm to have greater speedup than the copying algorithm. The copying algorithm only provides sequential access to the heap. If more than one thread tries to complete an operation simultaneously, only one can succeed and the rest must begin again after the heap pointer is changed. In our algorithm, multiple operations can be performed on the heap simultaneously. After the preliminary phase of an operation is completed, a new thread can begin an operation while the old thread performs the sifting phase of its operation. In certain pathological cases, it may take $\Omega(\log n)$ time before the next operation can begin. In practice, however, we expect the algorithm to behave more near the best case behavior of constant time per preliminary phase. Note that constant time per preliminary phase would yield a speedup within a constant of the optimal speedup of $\log n$.

7 Conclusions and Future Work

This algorithm shows it is possible for multiple asynchronous threads to simultaneously perform wait-free operations on the same copy of a heap data structure. An important property of the algorithm is that all suboperations are effectively atomic. The obvious next step is to code this algorithm and run it on a real machine to see how it performs in practice, particularly in comparison to the copying algorithm. Some tests should be run to determine whether the algorithm repeats less work due to contention and provides better speedup than the copying algorithm, as hypothesized above (Section 6).

The algorithm poses other interesting theoretical and practical questions. The lower bound given in Section 6 is not very satisfactory. It would be nice if we could show better bounds in special cases. Anderson and Woll [2] show that their Union-Find data structure algorithms perform much better if the threads can choose a request randomly from a large pool of outstanding requests. The same approach does not seem sufficient for a heap — the interference is caused more by the location in the tree where a thread works than by the data in the request itself, and this strategy will not change where the next key is positioned. A better strategy might be to choose a random leaf location when performing an insert operation, or to move the key of a random leaf to the root during a delete_min operation. With such a strategy, successive inserts would no longer necessarily operate in the same part of the tree, and it may be possible to make arguments about the work lost due to multiple insert operations similar to those made about delete_min operations in Section 6. If such a strategy is devised, we would also want to code it and see how it performs in practice.

Finally, the general approach of the algorithm could be useful in devising wait-free algorithms for other data structures. It seems likely that similar algorithms could be devised for objects that support only a small number of simple operations. For some objects, particularly those that are array-based or linear in nature, this approach could yield much more efficient wait-free implementations, since the obvious copying algorithm for these objects requires frequent copying of the entire structure, while an algorithm similar to ours would change only those portions of the object affected by a given operation. Even for objects for which a copying algorithm is efficient, one may prefer an algorithm that uses this approach because it suffers less contention and provides better speedup.

8 Acknowledgements

Richard Anderson originally posed this problem, and was helpful in listening to ideas and answering questions. Juan Alemany and Ed Felten discussed their paper and pointed out some similar work. Larry Ruzzo and Simon Kahan gave useful suggestions and comments.

References

[1] J. Alemany and E. W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, pages 124–134, Vancouver, B.C., Canada, Aug. 1992.

[2] R. J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. Technical Report 91-04-05, University of Washington, 1991. See also [3].

[3] R. J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, pages 370–380, New Orleans, LA, May 1991.

[4] G. Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 1993 ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, Velen, Germany, June 1993.

[5] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9(1):1–21, 1977.

[6] B. N. Bershad. Mutual exclusion for multiprocessors. Technical Report CMU-CS-91-116, Carnegie Mellon University, 1991.

[7] B. N. Bershad. Practical considerations for lock-free concurrent objects. Technical Report CMU-CS-91-183, Carnegie Mellon University, 1991.

[8] J. Biswas and J. C. Browne. Simultaneous update of priority structures. In *International Conference on Parallel Processing*, pages 124–131, 1987.

[9] R. Cole and O. Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, Santa Fe, NM, June 1989.

[10] R. Cole and O. Zajicek. The expected advantage of asynchrony. In *Proceedings of the 1990 ACM Symposium on Parallel Algorithms and Architectures*, pages 85–94, Crete, Greece, June 1990.

[11] R. Ford and J. Calhoun. Concurrency control mechanisms and the serializability of concurrent tree algorithms. In *Proceedings of the Third Annual ACM Symposium on Principles of Database Systems*, pages 51–60, 1984.

[12] P. B. Gibbons. A more practical PRAM model. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, Santa Fe, NM, June 1989.

[13] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science*, pages 8–21, Ann Arbor, MI, Oct. 1978. IEEE.

[14] M. Herlihy. A methodology for implementing highly concurrent data objects. In *Proceedings of the Second Annual ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 197–206, Mar. 1990.

[15] M. Herlihy. A methodology for implementing highly concurrent data objects. Technical Report CRL 91/10, DEC Cambridge Research Lab, Oct. 1991. See also [14].

[16] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[17] M. Herlihy and J. Wing. Axioms for concurrent objects. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 13–26, Munich, West Germany, Jan. 1987.

[18] D. W. Jones. Concurrent operations on priority queues. *Communications of the ACM*, 32(1):132–137, Jan. 1989.

[19] P. C. Kanellakis and A. Shvartsman. Efficient parallel algorithms can be made robust. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 211–222, Edmonton, Alberta, Canada, Aug. 1989.

[20] Z. M. Kedem, K. V. Palem, and P. G. Spirakis. Efficient robust parallel computations. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, pages 138–148, Baltimore, MD, May 1990.

[21] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[22] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(3):650–670, Dec. 1981.

[23] C. Martel, R. Subramonian, and A. Park. Asynchronous PRAMs are (almost) as good as synchronous PRAMs. In *Proceedings 31st Annual Symposium on Foundations of Computer Science*, pages 590–599, St. Louis, MO, Oct. 1990. IEEE.

[24] MIPS Computer Company. *The MIPS RISC architecture.*

[25] N. Nishimura. Asynchronous shared memory parallel computation. In *Proceedings of the 1990 ACM Symposium on Parallel Algorithms and Architectures*, pages 76–84, Crete, Greece, July 1990.

[26] Y. Sagiv. Concurrent operations on B-trees with overtaking. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Database Systems*, pages 28–37, Jan. 1985.